Programming Languages

2nd edition

Tucker and Noonan

Chapter 10
Function Implementation

In theory, there is no difference between theory and practice,
but not in practice.

Anonymous

Contents

10.1 Function Declaration and Call

Example Clite Program
Fig 10.1

```
int h, i;
void B(int w) {
        int j, k;
        i = 2*w;
        w = w+1;
}
void A(int x, int y) {
        bool i, j;
        B(h);
}
int main() {
        int a, b;
        h = 5; a = 3; b = 2;
        A(a, b);
}
```

10.1.1 Concrete Syntax

Functions and Globals (new elements $\underline{\text{underlined}}$)

$$\begin{aligned}
\mathit{Program} &\rightarrow \underline{\{\ \mathit{Type\ Identifier\ FunctionOrGlobal}\ \}} \\
&\qquad \mathit{MainFunction} \\
\mathit{Type} &\rightarrow \text{int} \mid \text{boolean} \mid \text{float} \mid \text{char} \mid \underline{\text{void}} \\
\underline{\mathit{FunctionOrGlobal}} &\rightarrow (\ \mathit{Parameters}\ )\ \{\ \mathit{Declarations\ Statements}\ \} \mid \\
&\qquad \mathit{Global} \\
\underline{\mathit{Parameters}} &\rightarrow \mathbf{[}\ \mathit{Parameter}\ \mathbf{\{}\ ,\ \mathit{Parameter}\ \mathbf{\}}\ \mathbf{]} \\
\underline{\mathit{Global}} &\rightarrow \mathbf{\{}\ ,\ \mathit{Identifier}\ \mathbf{\}}\ ; \\
\mathit{MainFunction} &\rightarrow \text{int main ( ) }\{\ \mathit{Declarations\ Statements}\ \}
\end{aligned}$$

Concrete Syntax (cont’d)

Function Calls (new elements $\underline{\text{underlined}}$)

$$
\begin{aligned}
\mathit{Statement} &\to \text{;} \mid \mathit{Block} \mid \mathit{Assignment} \mid \mathit{IfStatement} \mid \\
&\qquad \mathit{WhileStatement} \mid \underline{\mathit{CallStatement}} \mid \\
&\qquad \underline{\mathit{ReturnStatement}} \\
\underline{\mathit{CallStatement}} &\to \mathit{Call}\ \text{;} \\
\underline{\mathit{ReturnStatement}} &\to \text{return}\ \mathit{Expression}\ \text{;} \\
\mathit{Factor} &\to \mathit{Identifier} \mid \mathit{Literal} \mid (\ \mathit{Expression}\ ) \mid \underline{\mathit{Call}} \\
\underline{\mathit{Call}} &\to \mathit{Identifier}\ (\ \mathit{Arguments}\ ) \\
\underline{\mathit{Arguments}} &\to [\ \mathit{Expression}\ \{\ ,\ \mathit{Expression}\ \}\ ]
\end{aligned}
$$

10.1.2 Abstract Syntax

Program = *Declarations* globals; *Functions* functions

Functions = *Function**

Function = *Type* t; *String* id; *Declarations* params, locals;
Block body

Type = int | boolean | float | char | void

Statement = *Skip* | *Block* | *Assignment* | *Conditional* | *Loop* |
Call | *Return*

Call = *String* name; *Expressions* args

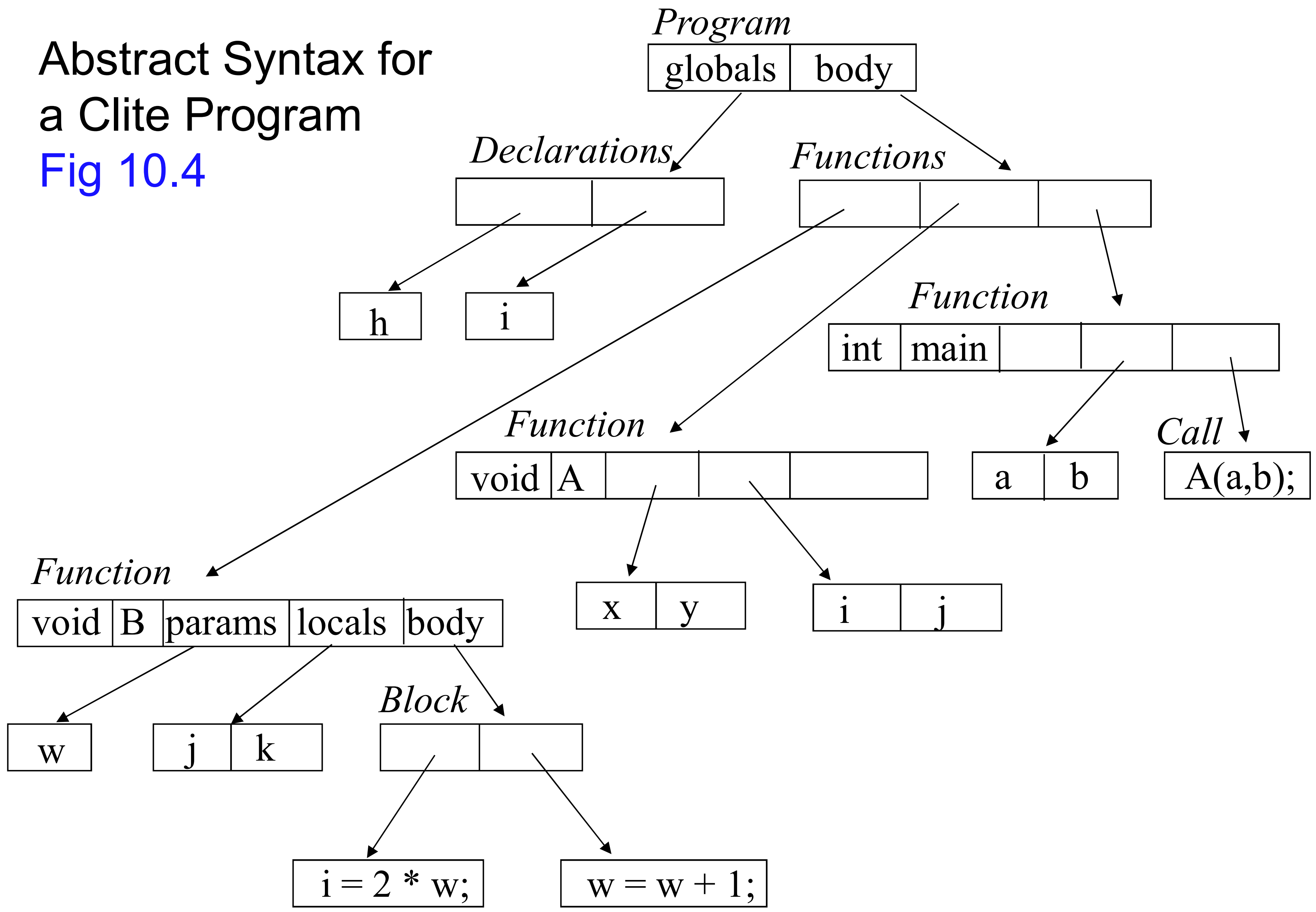
Expressions = *Expression**

Return = *Variable* target; *Expression* result

Expression = *Variable* | *Value* | *Binary* | *Unary* | *Call*

Abstract Syntax for a Clite Program

Fig 10.4



10.2 Completing the Clite Type System

Type Rule 10.1 Every function and global id must be unique.

E.g., h, i, A, B, and main are unique.

Type Rule 10.2 Every function's params and locals must have mutually unique id's.

E.g., function A's locals and params have id's x, y, i, and j.

Type Rule 10.3 Every statement in the body of each function must be valid with respect to the function's locals, params, and visible globals.

Note: Combine with type rules in Chapter 6 (type systems) for the different statement types.

Type Rules for Call and Return

Type Rule 10.4 A non-void function (except main) must have a *Return* statement, whose *Expression* must be the same type as the function.

Type Rule 10.5 A void function cannot have a *Return*.

Type Rule 10.6 Every *Call Statement* must identify a void function, and every *Call Expression* must identify a non-void function.

Type Rule 10.7 Every *Call* must have the same number of args as the number of params in the function it identifies. Each such arg must have the same type as its corresponding param, reading from left to right.

Type Rule 10.8 Every *Call* to a non-void function has the type of that function. The *Expression* in which the *Call* appears must be valid according to **Type Rules 6.5** and **6.6**.

Type Rules for Sample Program

Type Rule 10.4 No return statement appears in main.

Type Rule 10.5 No return statement appears in A or B.

Type Rule 10.6 The *Call Statement* A(a,b) identifies the function void A(int x, int y)

Type Rule 10.7 The *Call Statement* A(a,b) has two arguments, whose types (int) are the same as the parameters x and y.

Type Rule 10.8 is not applicable.

Example with Non-void Functions

Computing a
Fibonacci Number
Fig 10.5

```
int fibonacci (int n) {
    int fib0, fib1, temp, k;
    fib0 = 0; fib1 = 1; k = n;
    while (k > 0) {
       temp = fib0;
       fib0 = fib1;
       fib1 = fib0 + temp;
       k = k - 1;
    }
    return fib0;
}
int main () {
   int answer;
   answer = fibonacci(8);
}
```

Type Rules for Fibonacci program

Type Rule 10.4 return fib0; appears in int fibonacci (int n)

Type Rule 10.5 is not applicable

Type Rule 10.6 The *Call* fibonacci(8) identifies the non-void function int fibonacci (int n)

Type Rule 10.7 The *Call* fibonacci(8) has one argument, whose type (int) is the same as that of the parameter n.

Type Rule 10.8 The *Expression* in which the *Call* fibonacci(8) appears is valid.

10.3 Semantics of Call and Return

Meaning Rule 10.1 The meaning of a *Call c* to *Function f* has the following steps:

1. *Make an activation record and add f's params and locals to it.*
2. *Evaluate c's args and assign those values to f's corresponding params.*
3. *If f is non-void, add a result variable identical with f's name and type.*
4. *Push the activation record onto the run-time stack.*
5. *Interpret f's body.*
6. *Pop the activation record from the stack.*
7. *If f is non-void, return the value of the result variable to the Expression where c appears.*

Example Program Trace (Fig 10.6)

Calling	Returning	Visible State
main		<h,*undef*>, <i,*undef*>, <a,*undef*>, <b,*undef*>
A		<h,5>, <x,3>,<y,2>,<i,*undef*>, <j,*undef*>
B		<h,5>, <i,*undef*>,<w,5>,<j,*undef*>, <k,*undef*>
	B	<h,5>, <i,10>,<w,6>,<j,*undef*>, <k,*undef*>
	A	<h,5>, <x,3>,<y,2>,<i,*undef*>, <j,*undef*>
	main	<h,5>, <i,10>, <a,3>, <b,2>

10.3.1 Non-void Functions

Meaning Rule 10.2 The meaning of a *Return* is the result of assigning the value of the result *Expression* to the result variable.

Meaning Rule 10.3 The meaning of a *Block* is the aggregated meaning of its *Statements* when applied to the current state, up to and including the point where the first *Return* is encountered.

Note: The first Return encountered terminates the body of a called function.

Example Program Trace (Fig 10.7)

Calling	Returning	Visible State
main		<answer,*undef*>
fibonacci		<n,8>, <fib0,*undef*>,<fib1,*undef*>, <temp,*undef*>,<k,*undef*>, <fibonacci,*undef*>
	fibonacci	<n,8>, <fib0,21>,<fib1,34>, <temp,13>, <k,0>, <fibonacci,*undef*>
	main	<answer,21>

10.3.2 Side-Effects Revisited

- Side-effects can occur in Clite.
  - *E.g., in Figure 10.1, a call to* B *alters the value of global variable* i.
  - *If* B *had been non-void, this side-effect could have subtly affected an expression like* B(h)+i.
  - *In the semantics of Clite, evaluation of operands in a Binary is left-to-right. Thus, the result of* B(h)+i *is always the same, and always different from* i+B(h).

- Side-effects often create distance between mathematics and programming.
  - *E.g., Clite expressions with + are clearly not commutative, while in mathematics they are.*